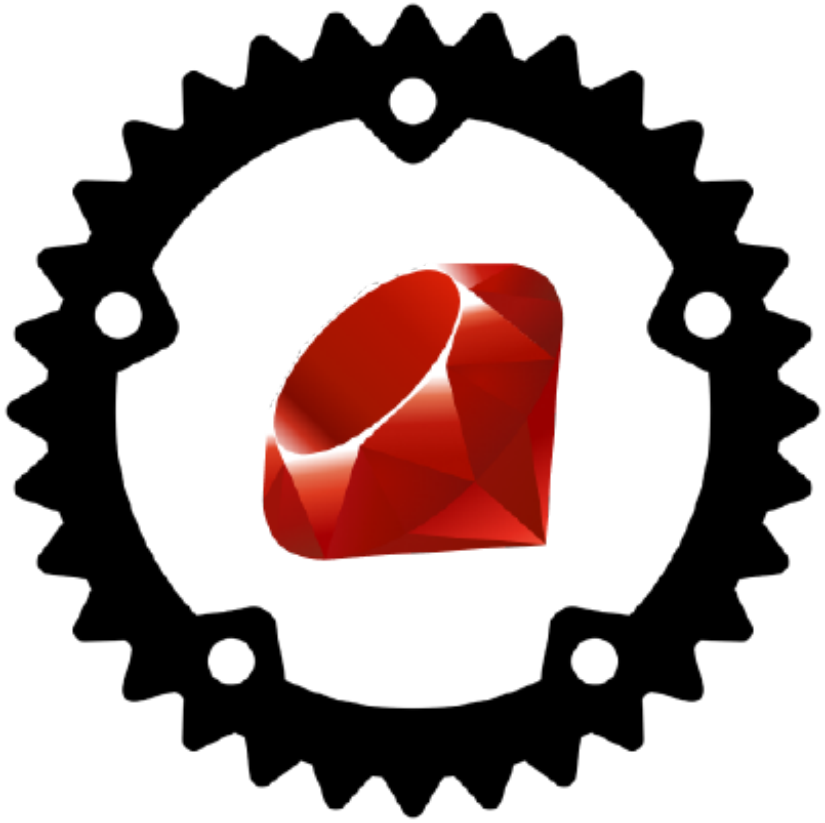

Ruby on Rust

An intro to writing native Ruby extensions in Rust.

You may be new to Rust, and that's OK! You'll find that Rust has many of the things you love about Ruby.

About Me

👋 I'm **@ianks**, and I work on the `liquid-perf` team. We are using Rust + WASM to improve the performance of Shopify's Liquid templating engine.

I am also the creator of the **`oxidize-rb`** open-source org. Our goal is to make writing native Ruby extensions in Rust easier than it would be in C.

What is a Ruby Extension?

For a native gem, we bypass this mechanism entirely and instead exposes native machine code to Ruby. In our native code, we can use the Ruby C API to interact with the Ruby VM.

```c
#include "hello.h"

VALUE hello(VALUE self) {
  return rb_str_new_cstr("hello");
}

void Init_hello(void) {
  rb_define_global_function("hello", hello, 0);
}
```

Why does it work with Rust and not other languages?

- Speaking in C, "lingua franca"
- Can compile functions with the C calling conventions
- Align items in memory in a way that C understands.
- Due to Rust's robust C FFI, you can code anything in Rust that you could with C.

What makes Rust a good choice for Ruby extensions?

- **Speed**: Rust is fast, comparable to C.
- **Memory Safety**: Rust is designed to prevent memory errors.
- **No GC**: Means we don't have to worry about 2 GCs running at the same time.
- **Ecosystem**: Rust has a large ecosystem of libraries and tools (cargo ~= bundler).
- **Familiarity**: Rust has many features that Ruby developers will be familiar with.

Use Cases

- **Performance**: You have identified a performance bottleneck that can't be solved in Ruby.
- **Complexity**: You have a complex native library in C that would benefit from using Rust enums, structs, and traits (yjit).
- **Bindings to Rust**: You want to make use of a Rust crate (wasmtime, cssparser, etc).
- **Bindings to C**: You want to make use of a C library (libxml, libcurl, etc), but want memory safety. You can use a Rust crate to wrap the C library (LLVM/inkwell, geos-rs, etc).

Learning Curve

The rumours are true, Rust has a steep learning curve. You will battle with the borrow checker, and fight with the compiler as you learn the language.

However, Rust errors typically happen at compile time, rather than segfaulting at runtime. This means that you can fix your errors quickly, and you can be confident that your code will work.

Shipping a Ruby Extension

When launching the `liquid-wasm` production verifier, I was expecting to spend days hunting down obscure segfaults.

I was shocked to fine that, besides unimplemented features, things just worked. In my experience, Rust gives you a "confidence to ship" that you don't get C.

Happy Path, or "How to write a Ruby extension in Rust"

1. magnus for to handle Ruby C API bindings.
    - Drop into rb-sys for low-level Ruby APIs.
2. cargo for dependency management.
3. Use the rb-sys gem for to make `cargo` work with Ruby (via `create_rust_makefile`).
4. rake-compiler for compiling the extensions (as you would with C)
5. *Not so distant future*: Support for `cargo` in RubyGems (done).

Demo Time

I made a small gem which implements a `String#reverse` using Rust and C. Let's compare and contrast:

- demos/ext/c
- demos/ext/rust_rbsys
- demos/ext/rust_magnus